Automatic implementations synthesis of secure protocols and attacks from abstract models

Camille Sivelle, Lorys Debbah, Maxime Puys, Pascal Lafourcade, Thibault Franco-Rondisson

▶ **To cite this version:**

Camille Sivelle, Lorys Debbah, Maxime Puys, Pascal Lafourcade, Thibault Franco-Rondisson. Automatic implementations synthesis of secure protocols and attacks from abstract models. Nordic Workshop on Secure IT Systems, 2022, Rekyavik, Iceland. pp.234-252, 10.1007/978-3-031-22295-5_13 . hal-03835049

HAL Id: hal-03835049
https://uca.hal.science/hal-03835049v1

Submitted on 31 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automatic Implementations Synthesis of Secure Protocols and Attacks from Abstract Models

Camille Sivelle[1], Lorys Debbah[1], Maxime Puys[1⋆[0000−0001−6127−9816]], Pascal Lafourcade[2[0000−0002−4459−511X]], and Thibault Franco-Rondisson[1]

[1] Univ. Grenoble Alpes, CEA, LETI, DSYS, Grenoble F-38000, France
Firstname.Name@cea.fr
[2] LIMOS, University Clermont Auvergne, CNRS UMR 6158, France
Pascal.Lafourcade@uca.fr

Abstract. Attack generation from an abstract model of a protocol is not an easy task. We present BIFROST (Bifrost Implements Formally Reliable prOtocols for Security and Trust), a tool that takes an abstract model of a cryptographic protocol and outputs an implementation in `C` of the protocol and either a proof in ProVerif that the protocol is safe or an implementation of the attack found. We use FS2PV, KaRaMeL, ProVerif and a dedicated parser to analyze the attack traces produced by ProVerif. If an attack is found then BIFROST automatically produces `C` code for each honest participant and for the intruder in order to mount the attack.

1 Introduction

The security of a communication protocol involves two different aspects: (i) the security of the protocol itself and (ii) the security of the cryptographic schemes involved. Several tools are now available to formally prove the intrinsic security of both protocols (like ProVerif [19], Scyther [23] or Tamarin [32]) and primitives (like CryptoVerif [20] or Easycrypt [9]) see [8] for a survey. However, another issue comes into play. Security flaws often appear when implementing code for a given protocol, even for a proven secure one. In this case, the attack does not rely on an intrinsic flaw of the protocol, but involves vulnerabilities related to the code design or even from the programming language. It also happens too often that the implementation is a slightly different protocol than the one proven (for instance the order of the content of an encrypted message is changed) and thus have the formal proof becoming meaningless. In 2014, the Heartbleed [27] attack over SSL/TLSv1.0 whose feasibility had been formally proven in [16] is an example of an attack targeting the implementation. We can also mention a famous attack of the SSH protocol in Debian Linux distributions where the generation of a fresh nonce was wrongly implemented [5]. Hence implementing a secure protocol is a sensitive task and every detail is important.

⋆ Corresponding author: Maxime PUYS, CEA-Leti, 17 rue des Martyrs, 38054 GRENOBLE CEDEX 9, France, Maxime.Puys@cea.fr

Contributions: We propose an automatic tool-chain named BIFROST (Bifrost Implements Formally Reliable prOtocols for Security and Trust), that takes as input a cryptographic protocol modeled in F*, we call this input a *protocol model* in the rest of the paper, and combines the following existing tools: FS2PV [17], KaRaMeL [38] (formerly known as KreMLin) and ProVerif [19]. BIFROST produces:

1. An implementation in `C` from a protocol modeled in F*.
2. If the protocol is safe then a proof in ProVerif is produced.
3. If ProVerif finds a flaw then an implementation in `C` of the attack is given.

BIFROST supports several standard cryptographic primitives that correspond to those supported by ProVerif. We are able to use several symmetric encryption schemes, public key encryption schemes, signatures schemes and hash functions. All these primitives are wrapped around widely-known cryptographic primitive libraries such as MbedTLS. BIFROST unifies their APIs and make them directly compatible with verification tools such ProVerif. BIFROST has been successfully tested on the famous Needham-Schroeder [34] and Otway-Rees [36] protocols, alongside a MAC based password protocol taken from [17].

Formal verification tools need to over-approximate an attacker's capabilities in order to be sure that a protocol is secure in regards to a given property. Therefore, the attacks generated by such tools can sometimes not be feasible in practice. If the protocol was not found secure, the attack implementation generated by BIFROST can be played along with the protocol implementation previously generated. If the execution of the attack found by formal verification succeeds, it proves its feasibility in a practical context. This can give an automatic confirmation that the protocol is not secure, rather than manually implementing the attack.

Related Work: Several tools can be related to BIFROST. We classify them in three categories: (i) generic code verification tools, (ii) cryptographic protocol verification tools, (iii) tools specifically designed to generate code from verifiable protocol models. Generic code verification and cryptographic protocol verification tools are not directly related works to BIFROST, but they are related to the basic blocks used in the approach. Thus, we will mainly mention the one used within BIFROST and their main competitors.

Generic code verification tools: They have been developed for several tens of years. Among many other reference tools we can list:

- The B method [6] is a formal method software development framework proposed by Abrial et al. in 1996 based on set theory and first order logic in order to write and check code specifications. The goal is to both check consistency of specifications and code.
- Frama-C [24] is a tool developed in 2012 by Cuoq et al. that performs static analysis on `C` programs. Various analyses are supported such as dead code deletion, value analysis or weakest-precondition calculus.

– F* [40] is a general-purpose functional programming language designed by Swamy et al. in 2013 which allows to specify properties alongside code. Then, various analyses can be performed on the code such as dependent types, monadic effects, refinement types, and a weakest precondition calculus.

We use F* models in BIFROST, due to their compatibility with the other bricks used in our toolchain.

Cryptographic protocol verification tools: They have been developed since 1995, when G. Lowe found an attack on the formerly proven Needham-Schroeder protocol [31]. Such tools often implement the Dolev-Yao [26] intruder model and check all possible actions for an attacker interacting with multiple sessions of a given protocol in parallel to verify security properties such as secrecy or authentication. Multiple tools have been introduced since 1995 [8] and benchmarked [30,8]. Among them we can list:

– Tamarin [32] is a security protocol prover designed by Meier et al. since 2013. Tamarin is able to handle an unbounded number of sessions. Protocols are specified as multi-set rewriting systems with respect to temporal first-order properties. It relies on Maude [28] and supports equational theories such as Diffie-Hellman.
– DY* [14] is a tool developed by Bhargavan et al. in 2021. It is an implementation of the Dolev-Yao intruder model in F* and allows security properties verification on a protocol taking advantage of the internal F* prover. It is however currently unable to produce attack traces.
– ProVerif [19] is developed by Blanchet et al. since 2001 and relies on Horn clause analysis to check an unbounded number of sessions.

We chose ProVerif in our tool chain, since it is stable and that several bricks of our approach are compatible with this well established tool.

Cryptographic protocol code generation tools: There exist some tools allowing to generate code from verifiable protocol models, such as BIFROST.

– Spi2Java [37] is a framework proposed in 2004 that automatizes the generation of Java implementations from protocols described in spi calculus, an extension of pi-calculus. This method allows for formal verification of security properties through translation of the spi-calculus specifications to a format that can be verified by ProVerif prior to code generation.
– In 1993 and in 2009, Bieber et al. [18] and Benaissa et al. [12] respectively proposed an approach to analyze the security of cryptographic protocols using the Event-B framework. To the best of our knowledge, they partly implement the Dolev-Yao model as a library for the internal verifier of Event-B, allowing them to specify lemmas describing security properties to be proven such as secrecy and authentication. It is however unclear if their approach is able to find an attack trace. As their framework relies on Event-B, specifications can be refined into `C`-like code.

– In 2009, Bhargavan et al. [15] proposed a compiler allowing to translate protocols modeled in some ad-hoc language into ML-like implementations. They provide various security verifications through a custom type-checker [13] which performs security verifications similar to ProVerif and Tamarin.
– AnBx [33] is an IDE developed in 2015 by Modesti. It extends the Alice & Bob (AnB) protocol model making it compatible with OFMC [10], a protocol verifier such as ProVerif. After verification, the protocol model can be translated into Java.
– Jasmin [7] is a cryptographic primitive verification tool, developed by Almeida et al. in 2017. It takes a primitive model as input, written in a specific language, and checks it against memory flaws or cache timing attacks. The model is then translated in a subset of the `C` language. Yet, it is worth noting that even if Jasmin shares resemblance with the frameworks described above, it only applies to cryptographic primitives rather than to protocols, which are complementary.

Finally, several previous works mention that they perform *protocol synthesis*. However, if their works share resemblance with ours, this terminology should not be confused with *protocol implementation synthesis* which aims at automatically generating executable protocol implementations. For instance, Bellare et al. [11] and Katz et al. [29] synthesize protocol models resistant to active intruders from protocol models resistant to passive intruders in the context of authenticated group key exchange. Cortier et al. [22] translate a single-session protocol into a multi-session protocol secure against a Dolev-Yao intruder. Sprenger [39] et al. rely on Isabelle/HOL [35] to write secure-by-design protocol models. In 2008, Bhargavan et al. [17] synthesize ProVerif models from F# protocol implementations in a tool called FS2PV. We are using FS2PV in BIFROST because it is compatible with ProVerif and it will help us in our goal to generate attack implementations in `C`.

Overall, only cryptographic protocol code generation tools are direct competitors to BIFROST. All other presented works are related to internal tools used within BIFROST (e.g., cryptographic protocols verification tools). Moreover, to the best of our knowledge, if all tools mentioned above are able to faithfully translate a protocol from a provable model into a programming language (with their own limitations), none of them are dealing with attacks found by the verification tools.

Outline: In Section 2, we introduce the BIFROST framework. In Section 3, we delve into the technical challenges regarding automatic code generation from protocol models and explaining the inner-workings of BIFROST. In Sections 4 and 5, we respectively present the cryptographic primitives supported and how we automatically generate code for attacks found. In Section 6 we give a detailed example of the approach on the Needham-Schroeder protocol which will be part of technical report of this paper and in the manual of BIFROST. Finally, we conclude in Section 7.

2 Overview of BIFROST

Our aim is, with the same input file, to be able to generate `C` code with KaRaMeL and a π-calculus file for ProVerif. For this, we use a subset of F* that is compatible both with the subset of F# that is used by FS2PV, and also with `low*` (the subset of F* that can be compiled into `C`) in order to be able to use KaRaMeL. In the rest of the paper, we denote this subset $F^{\$}$. When it is clear from the context we also use F*. In Figure 1, we describe the BIFROST approach. From a cryptographic protocol model (1) the user needs to write an $F^{\$}$ file. In step (2), we generate a π-calculus model using FS2PV, which can then be verified by ProVerif in step (3) with respect to the security properties described in a `.query` file. If the protocol is safe and ProVerif proves the security of the requested properties in step (4), we apply KaRaMeL in step (5) to the initial model in $F^{\$}$ to generate `C` code corresponding to the implementation of the roles of each participant in the protocol (6), thus bridging the gap between the formal verification of the protocol and its implementation. If the protocol is not safe it means that ProVerif found an attack in step (7), then we parse the ProVerif attack trace using a tool we have developed (8) to generate the corresponding $F^{\$}$ code. We apply KaRaMeL to the obtained $F^{\$}$ files to automatically generate the `C` code implementing the role of the attacker in the protocol.

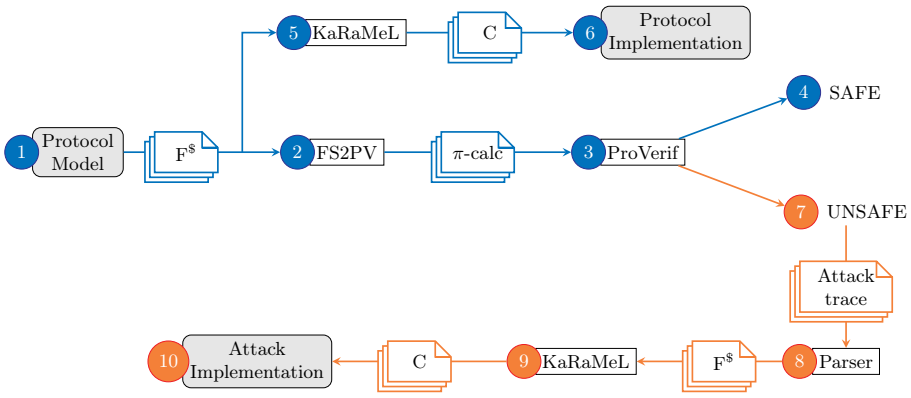

Fig. 1: The BIFROST toolchain.

In the following, we describe in Section 3 some of the challenges for transforming abstract models into a practical implementation. We then present, in Section 4, the different cryptographic primitives that are available in BIFROST and how we integrate them to allow crypto-agility. Finally, we show, in Section 5, how we manage to generate attack implementations from ProVerif traces.

3 From Abstract Model to Implementation

The translation from an abstract model to a concrete implementation brings several challenges. The F$^\$$ model which serves as an input for our toolchain relies on a set of functions for cryptographic primitives, network operations and data manipulation, which are all currently imposed by FS2PV. One important point is that FS2PV only uses an abstract definition of these function to translate the input F$^\$$ protocol description into a ProVerif model. However, when generating the `C` implementation of the protocol, all these functions must also be defined with their proper implementation. Moreover, their implementation must fit their purpose in the real world. For instance, if the `Net.send` function is internal to FS2PV for its analysis at the abstract model phase, it must actually be sending packets on a network when called by real code. Some abstract functions can have multiple implementations depending on user choices. For instance, a `Crypto.symenc` function can be translated into AES or ChaCha encryption while `Net.send` can translate to a TCP/IP send or a LoraWan send depending on context. On the other hand, some functions defined and called within the abstract model do not serve any purpose in the implementation (such as π-calculus' $fork$).

BIFROST libraries. To this end, we propose a series of `C` libraries `Crypto`, `Net`, `Data` and `Pi`, implementing all the necessary functions to link and run the `C` code produced by KaRaMeL from our model. Figure 2 describes the translation of the different libraries of abstract functions into their implementations.

- `Data` contains several functions necessary for data types manipulation. In FS2PV representation, the principal data type is the Byte, which can represent variables of various size and nature, as it could be a nonce as well as a key, or even a concatenation of different Bytes. This is problematic in `C`, as those can hardly be represented by the same data structures and might introduce some type flaw attacks, as the one existing on the fixed version of Needham-Schroeder by G. Lowe, where a confusion between identity and nonces allows an intruder to mount an attack [21]. The solution we proposed in our `Data` library is to define the Byte as a C structure, composed of an integer representing the subtype of the object, an integer representing the size of the object and an union of the C structures corresponding to each possible subtype (nonce for example).
- `Crypto` gathers all the functions relative to the cryptographic operations in the protocol. Most of its functions are meant to be crypto-agile and thus are algorithm agnostic wrappers to specific primitives such as RSA or AES.
- The `Net` library of BIFROST includes basic network operations. As of now, they are implemented on either TCP/IP sockets or low level UART connections. However, supporting various protocols is possible with BIFROST but it is important to ensure the compatibility with the network model of ProVerif.
- Finally, the `Pi` library relates to internal π-calculus functions used by ProVerif for its analysis (mainly for process scheduling) and does not have any real purpose in real life. Thus, this module does not need to be translated into `C`.

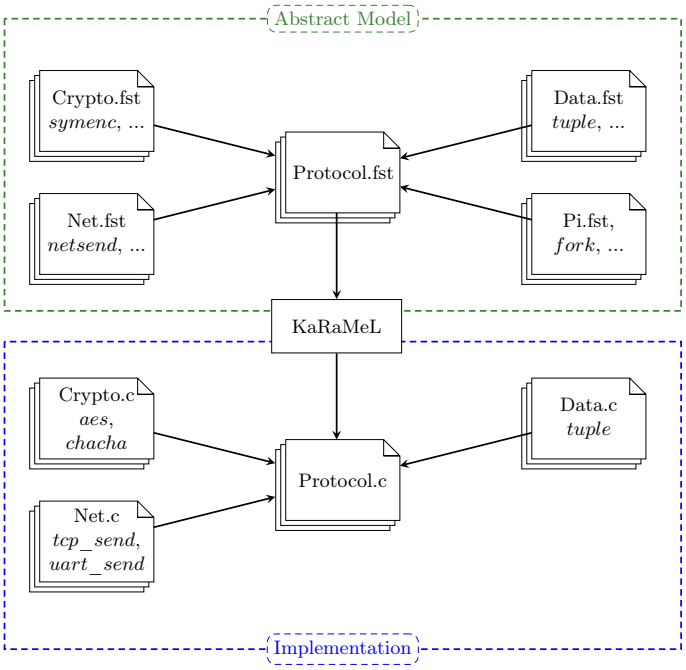

Fig. 2: Abstract functions libraries translated into `C` code.

Keys management. Another difference between the protocol model and the role implementation is the way that keys are managed. When verifying a protocol with ProVerif, private keys are often modeled as terms freshly generated at the beginning of the protocol, public keys being derived from these terms and published on a public channel. This is sufficient to consider that each participant knows a given public key in the formal model. However, this traditional setup for verification, consisting in broadcasting keys or supposing that participants already know each other's public keys beforehand is more complicated to pull off on real hosts. Within BIFROST, we use the following key management to reconcile both formal verification and implementation usability (it can also apply to any supposedly pre-shared keys): two distinct main functions are defined for the formal verification and the implementation generation. One is required by FS2PV in which each participant role is called with keys created by abstract methods from FS2PV. This file is then translated as the main *process* clause used by ProVerif to setup roles instances. On the other hand, KaRaMeL produces `C` code for every participant, plausibly running on different systems, and each of them requires its main function. Thus, the role of each participant is described in a $F^{\$}$ file containing a role function, which takes in argument the keys, and a main function, in which the keys are loaded from a `.pem` file. This choice presupposes that the keys are generated and stored on each system executing a participant's role beforehand, which is common for embedded devices.

4 Cryptographic Primitives

To allow crypto-agility by letting the user choose a cryptographic algorithm among different options, we decided to handle all the following schemes and to implement several algorithms for each of them.

- **Symmetric encryption** uses a key shared by both participants to encrypt and decrypt data. Symmetric keys are often smaller than asymmetric keys which allows communication to be fast. We support three algorithms for symmetric encryption: AES-128/192/256 in CBC mode, Blowfish in CBC mode and Chacha20.
- **Message Authentication Code** uses symmetric keys to check the integrity and authenticity of data. We support the following four algorithms: AES with CMAC, Poly1305-AES, Chacha20-Poly1305 and HMAC.
- **Hashing** has a lot of applications in cryptography. It can be used to compute fingerprints for example. We support two hashing algorithms: SHA2 and RIPEMD160.
- **Asymmetric encryption** uses a public key, private key pair to encrypt and decrypt. It is often used to transfer symmetric keys. We support two algorithms for asymmetric encryption: RSA-OAEP and RSA-PKCS1-v1.5.
- **Asymmetric signature** uses a public key, private key pair to check for the integrity and the authenticity of data. We support three algorithms for asymmetric signature: RSA-PKCS1-v1.5, RSA-PSS and ECDSA.

Cryptographic primitives management in BIFROST relies on MbedTLS[1], a `C` library developed by ARMmbed. It implements the TLS [25] protocol and required cryptographic primitives. MbedTLS is designed to fit on small embedded systems. However, one of the key concerns of BIFROST is to allow for crypto-agility, which would in our case translate as the possibility for the toolchain to handle both different algorithms and different cryptographic libraries, such as OpenSSL [3] or OpenQuantumSafe [2], in order to fit as easily to existing code base. In practice, any cryptographic operation requires different MbedTLS function calls (e.g., to set the seed or initialize the context before encrypting). As all these atomic function calls are not defined within FS2PV and ProVerif, granularity differs with MbedTLS. Thus, we created a `C` library composed of functions that would act as wrappers above MbedTLS's functions. More precisely, we wanted those wrappers to encompass every intermediate function of MbedTLS (or another library) to provide a more generic function to the user, which would also fit the granularity of protocol models. The use of this library also allows the user to choose the primitives to use for a given protocol.

Different options allow us to choose which primitive to use. The choice of which primitive should be used is currently left to the user of BIFROST (which is the protocol modeler and not the final user). This choice is partly motivated by the context of embedded systems where not all primitives are supported by chip vendors and some freedom in the choice can be important. Yet, letting any user choose their cryptographic primitives will often lead to vulnerable systems. While

leaving room for freedom of choice, we intend to support cryptographic suites[3] which will automatically include a combination of validated cryptographic primitives with their proper configuration (e.g., ECDHE_PSK_WITH_AES_128_GCM_SHA256). By defining the right macros, preprocessor will comment out code for unused primitives. In a similar way, the user can choose the size of its keys with preprocessor directives, the idea being to give the user maximum control over the parameters for each primitive. This control is the one that MbedTLS provides so we did not restrict the options provided by the library but we eased the way to select them. This allows a user to custom the protocol by choosing the cryptographic primitives' parameters he wishes to use.

5 Attack Generation

The generation of the attack implementation is based on the output printed by ProVerif in case an attack is found. This output corresponds to an attack trace in applied π-calculus which describes the steps performed by the intruder to violate the specified security property. As already presented in Figure 1, the approach we chose for this part was to first parse the attack trace from ProVerif's output in order to obtain an abstract syntax tree (8), and to use that tree to generate the $F^{\$}$ file implementing the actions described in the trace. We then apply KaRaMeL (9) to that $F^{\$}$ file to generate the `C` implementation of the attacker's role (10). We motivate the choice of generating $F^{\$}$ code and using KaRaMeL rather than generating `C` code directly from the syntax tree for the coherency it guarantees with the `C` code generated from the protocol model with KaRaMeL. To our knowledge, there is no tool or framework allowing to generate executable code for an attack found by a protocol verification tool. The generated C code can then be compiled and executed alongside a normal protocol session (or more if needed) in order to play the attack.

ProVerif Attack Trace Parsing: To parse the attack traces from the ProVerif outputs, we have chosen to proceed in Python, using the Lark module [4]. Lark is a parsing library able to parse any context-free grammar, using an advanced grammar language based on EBNF, a metalanguage that allows to describe the syntactic rules of programming languages. The first step is therefore to determine the grammar of ProVerif outputs and to describe it in EBNF language. Globally, an attack trace is made up of a succession of lines that can be considered as instructions. We consider four different types of instructions in ProVerif's language: *in*, *out*, *new* and *event*.

```
1 in(c,(Bob,M_1)) with M_1 = pk(skB_1) at {5}
2 out(c,M_6) with M_6 = aenc(n_3,pk(skB_1)) at {12}
```

Listing 1.1: Example of in/out instructions.

[3] `https://ciphersuite.info/cs/`

The *in* and *out* instructions aim to send and receive messages from a channel and have the same grammar. The elements that we want to retrieve in our syntax tree are:

1. their two arguments (for example c and M_6 for the *out* of Listing 1.1),
2. the equality given after the with keyword (for instance $M\_1 = pk(skB\_1)$ for the *in*),
3. the line number in the process (5 or 12).

In Listing 1.1, c, M_6, and M_1 are variable or channel names, $pk(skB\_1)$ or $aenc(n\_3, pk(skB\_1))$ are functions applied these variables. As terms can be functions applied multiple times to another term, we define values as either ground terms or function results and line numbers correspond to natural numbers. Then, we can define the *in* and *out* instructions as in Listing 1.2.

```
1 out: "out" "(" val "," val ")" "with" val "=" val "at" "{" line "}"
2 in: "in" "(" val "," val ")" "with" val "=" val "at" "{" line "}"
```

Listing 1.2: Grammar for in/out instructions.

We also define similar rules for *new* and *event* keywords. The *new* instruction allows a participant or the intruder to generate a fresh term. On the other hand, *event* does not have any effect on the protocol but allows to place markers in the trace which can be used for reachability properties. Their form is shown in Listing 1.3.

```
1 new n_1 creating n_3 at {21}
2 event endB(A,pk(skA_1),Bob,pk(skB_1),n_2) at {25}
```

Listing 1.3: Example of new/event instructions.

Similarly, these instructions are translated in EBNF as shown in Listing 1.4.

```
1 new: "new" val "creating" val "at" "{" line "}"
2 event: "event" val "at" "{" line "}" "(goal)"?
```

Listing 1.4: Grammar for new/event instructions.

Generating F$ Code: From the syntax tree obtained, we write a program that generates a code in F$. Processing will differ according to the type of instruction. An *out* corresponds for example to a sending by one of the hosts defined in the protocol. As the code generated at this step corresponds to the behavior of the intruder, a call to F$ *Net_recv* function is performed on the channel indicated by the variable stored in the AST. Similarly, an *in* means that the intruder must send the given message on the channel, leading to a call to *Net_send*. When the declaration in the *with* statement involves functions, for example $M\_6 = aenc(n\_3, pk(skB\_1))$, we translate these into the corresponding functions in our libraries. The events could be ignored since they do not bring anything to the attack in itself. We therefore simply use a logging system when an event is raised. The case of the *new* requires a little more work because ProVerif does not distinguish the generation of nonces from the generation of encryption keys.

However, the functions that are assimilated in F$^{\$}$ are not the same. To remedy this problem, we use a preprocessing function that goes through the tree a first time to determine on one hand if the generation of each variable is done by the intruder, and on the other hand, the type of this variable. The latter is given by the functions that are applied to it after creation. The algorithm used for the generation of the code can therefore be summarized as shown in Listing 1.5.

```
ast_to_fdollar(tree):
  initialize the output string to ""

  # Preprocessing
  for each instruction:
    if it is a new:
      store the type of the variable declared
  output all the declarations with their correct types

  # Main processing
  for each instruction:
    determine the type using the first child node of the tree
    if it is of type "in":
      add to output a "let ... = ..." defining the variable
      add to output a Net_send(...,...)
    if it is of type "out":
      add to output a "let ... = ..." defining the variable
      add to output a Net_recv(...,...)
    if it is of type "event":
      add to output a log(...,...)
    if it is of type "new":
      do nothing (already processed during the preprocessing)
  return output
```

Listing 1.5: F$^{\$}$ code generation.

6 An Example: the Needham-Schroeder Protocol

We show how BIFROST can be applied on the well-known Needham-Schroeder protocol. We first describe this protocol in Section 6.1, then we show in Section 6.2 how this protocol is modeled in F$^{\$}$ and it is translated into `C` code using BIFROST. Finally, we detail in Section 6.3 the generation of the implementation of an attack on this protocol from the attack trace found by ProVerif.

6.1 The Needham-Schroeder Protocol

The Needham-Schroeder [34] protocol is a mutual authentication protocol involving two parties A and B. They wish to agree on a shared value that they will use to secure further communications. In this protocol, messages are sent on an insecure channel. N_A, N_B are nonces, pk_A, pk_B are public keys and $(m)_{pk_B}$

symbolizes the public encryption of the message m. The first message $(A, N_A)_{pk_B}$ is used to initiate a new session between A and B. The second message is used by A to authenticate B and the third one is used by B to authenticate A. Nonces are also used to prevent replay attacks.

$$\begin{aligned} A &\longrightarrow B : (A, N_A)_{pk_B} \\ B &\longrightarrow A : (N_A, N_B)_{pkA} \\ A &\longrightarrow B : (N_B)_{pk_B} \end{aligned}$$

In 1996 G. Lowe [31] found a famous attack on this protocol. This attack assumes that a dishonest agent I impersonates the honest agent B in the previous protocol, leading to a man-in-the-middle attack.

$$\begin{aligned} A &\longrightarrow I : (A, N_A)_{pk_I} \\ I &\longrightarrow B : (A, N_A)_{pkB} \\ B &\longrightarrow I : (N_A, N_B)_{pk_A} \\ I &\longrightarrow A : (N_A, N_B)_{pk_A} \\ A &\longrightarrow I : (N_B)_{pk_I} \\ I &\longrightarrow B : (N_B)_{pk_B} \end{aligned}$$

With this attack, the intruder obtains both N_A and N_B which allows him to derive any secret based on those two nonces. He also impersonates A when speaking to B. This vulnerability was fixed by Gavin Lowe [31] by modifying only one message in the Needham-Schroeder-Lowe (NSL) protocol:

$$\begin{aligned} A &\longrightarrow B : (A, N_A)_{pk_B} \\ B &\longrightarrow A : (N_A, N_B, B)_{pkA} \\ A &\longrightarrow B : (N_B)_{pk_B} \end{aligned}$$

One can see that with this variant, the intruder is not able to get the nonce N_B. Indeed, as the intruder tries to perform the attack, A will cipher with pk_I as she is willingly talking with the intruder. However, upon reception of the second message containing the identity of B, A will realize she is not talking to I as she was willing to initially and will not send the last message, thus preventing I from getting the last nonce.

6.2 From Protocol Model to Implementation

The code 1.6 is the role of A in the Needham-Schroeder protocol in $F^{\$}$, and the code 1.7 is the protocol implementation in `C` generated by KaRaMeL are given in Figure 3. The code for B is similar. We can see that in $F^{\$}$ code 1.6, we start by establishing the communication channel `10.0.0.2`, port 80 (which we will use for the IP address of B). Then a pair of keys is generated and public keys

```
let roleA host : Int32.t =
 let url="10.0.0.2:80" in
 let c=connect(url) in
 let skA=(rsa_keygen()) in
 let pkA=(rsa_pub skA) in

 let x0=Net.send c (concat
     (keytobytes pkA) host
    ) in

 let (pkb,dest)=(iconcat(
    Net.recv c)) in

 // A -> B: (A,nA)_pkB
 let pk=asympubkey(pkb) in
  let nA=mkNonce() in
  log tr (BeginA(host,dest
    ,nA));
  let x1=Net.send c (
    rsa_encrypt pk (concat
     host nA)) in

  // B -> A : (nA, nB)_pkA
  let m=(rsa_decrypt skA (
    Net.recv c)) in

   let (nA1, nB)=iconcat m
     in
  if nA1=nA then
  (
  // A -> B : (nB)_pkB
  let x2=(Net.send c (
    rsa_encrypt pk nB)) in
  let z=(close c) in 0l
  )
  else let z=(close c) in
    0l
```

Listing 1.6: Role of A in F$^{\$}$.

```
int32_t roleA(Data_bytes host) {
  Prims_string url ="10.0.0.2:80"
    ;
  Net_conn c = Net_connect(url);
  Crypto_key skA =
    Crypto_rsa_keygen();
  Crypto_key pkA = Crypto_rsa_pub
    (skA);
  Prims_int x0 = Net_send(c,
    Data_concat(Crypto_keytobytes
    (pkA), host));
  K___Data_bytes_Data_bytes scrut
    = Data_iconcat(Net_recv(c));
  Data_bytes pkb = scrut.fst;
  Crypto_key pk =
    Crypto_asympubkey(pkb);
  Data_bytes nA = Crypto_mkNonce
    ();
  Prims_int x1 = Net_send(c,
    Crypto_rsa_encrypt(pk,
    Data_concat(host, nA)));
  Data_bytes m =
    Crypto_rsa_decrypt(skA,
    Net_recv(c));
  K___Data_bytes_Data_bytes
    scrut0 = Data_iconcat(m);
  Data_bytes nA1 = scrut0.fst;
  Data_bytes nB = scrut0.snd;
  if (__eq__Data_bytes(nA1, nA))
  {
    Prims_int x2 = Net_send(c,
    Crypto_rsa_encrypt(pk, nB));
    Prims_int z = Net_close(c);
    return (int32_t)0;
  }
  else
  {
    Prims_int z = Net_close(c);
    return (int32_t)0;
  }
}
```

Listing 1.7: Role of A generated by KaRaMeL.

Fig. 3: Input code of A's role in F$^{\$}$ and generated `C` code.

are published on the network. A listens for the public key of B [4]. After this initialization phase, the protocol can start and A generates a nonce and sends it and her name encrypted with the public key of B according to the first step of the protocol. Then A waits for the answer of B. Once B's response is received, A checks the correspondence between the nonce sent in the first message and the one received from B. If the nonces match them A confirms to B that she has received N_B by sending it back to B encrypted by B's public key.

The `C` code produced by BIFROST is given in the code 1.7. It follows exactly the same steps and use all libraries proposed by the tool. Moreover, we argue that even for an automatically generated code. It stays fairly understandable and possible to analyze with `C` static analyzers such as Frama-C or CPPcheck.

6.3 Attacker Implementation Generation

The F$^{\$}$ code of the role of A displayed in Listing 1.6 is translated into *pi*-calculus by FS2PV and can be analyzed by ProVerif [5]. To do so, we need to provide a *query* (i.e., a security property) for ProVerif to verify. In this example, this query is shown in Listing 1.8 and requires the last message received by B from A to actually be sent by A for B earlier, ensuring authentication of A to B on N_B.

```
query ev:Ev(BMessageB(a,b,nb)) ==> ev:Ev(AMessageA(a,b,nb)).
```

Listing 1.8: Query used by ProVerif

Using this query, ProVerif is able to find an attack. The trace is also quite long and we chose to narrow down the output to what is shown in Listing 1.9.

```
new T55 creating T55_1 at {90}
new T53 creating T53_1 at {92}
new T51 creating T51_1 at {94}
out(NethttpChan, CryptoAsymPrivKey(DataFresh(M))) with M =
    T51_1 at {96}
out(NethttpChan, CryptoAsymPubKey(DataBin(M_1))) with M_1 =
    DataPK(DataFresh(T53_1)) at {97}
event Ev(BStart(DataUtf8(SBobS()),DataUtf8(SAliceS()))) at
    {128}
event Ev(AStart(DataUtf8(SAliceS()),DataUtf8(SIntruderS()))
    at {100}
new T49 creating T49_1 at {102}
out(NethttpChan, DataBin(M_2)) with M_2 = DataAsymEncrypt(
    DataBin(DataPK(DataFresh(T51_1))),DataConcat(DataUtf8(
    SAliceS()),DataFresh(T49_1))) at {104}
in(NethttpChan,[...]) = DataAsymEncrypt(DataBin(DataPK(
    DataFresh(T53_1))),DataConcat(DataUtf8(SAliceS()),
    DataFresh(T49_1))) at {130}
new T29 creating T29_1 at {135}
```

[4] This is a common modeling practice in protocol verification, allowing the intruder to choose who A is going to talk to.

[5] As this code is really long, we will not show it in this article.

```
12 out(NethttpChan, DataBin(M_3)) with M_3 = DataAsymEncrypt(
       DataBin(DataPK(DataFresh(T55_1))),DataConcat(DataFresh(
       T49_1),DataFresh(T29_1))) at {137}
13 in(NethttpChan, DataBin(M_3)) with M_3 = DataAsymEncrypt(
       DataBin(DataPK(DataFresh(T55_1))),DataConcat(DataFresh(
       T49_1),DataFresh(T29_1))) at {105}
14 out(NethttpChan, DataBin(M_4)) with M_4 = DataAsymEncrypt(
       DataBin(DataPK(DataFresh(T51_1))),DataFresh(T29_1)) at
       {111}
15 in(NethttpChan,[...]) = DataAsymEncrypt(DataBin(DataPK(
       DataFresh(T53_1))),DataFresh(T29_1)) at {138}
```

Listing 1.9: Attack trace found by ProVerif.

The attack found by using ProVerif is actually the same as the one discovered by Lowe and presented in Section 6.1. It can be read as the following (with lines 1-5 prior to the roles of A and B): Line 1 : creation of sk_A; Line 2 : creation of sk_B; Line 3 : creation of sk_I; Line 4 : share sk_I for the intruder; Line 5 : share pk_B for the intruder; Line 6-7 : start event from B and A Line 8 : creation of nonce N_A; Line 9 : A sends $(A, N_A)_{pk_I}$; Line 10 : B receives $(A, N_A)_{pkB}$. This message is built by the intruder using the message from A. Line 11 : creation of nonce N_B; Line 12 : B sends $(N_A, N_B)_{pk_A}$; Line 13 : A receives $(N_A, N_B)_{pk_A}$; Line 14 : A sends $(N_B)_{pk_I}$; Line 15 : B receives $(N_B)_{pk_B}$. This message is built by the intruder using the message from A.

The $F^{\$}$ representation of the role of the intruder generated by the Python parser can be found in Listing 1.10. The reader may take note that it follows the steps of the attack described above. The `C` implementation is generated by KaRaMeL, similarly to the roles of A and B, and after compilation it can be executed along with the two roles in order to play the attack. Finally, when we add Lowe's correction to the protocol model (described in Section 6.1), we can indeed see that ProVerif declares the protocol safe for the given properties.

```
1  let roleI skI pkB adr1 adr2: Int32.t =
2      let c2 = connect adr2 in
3      let c1 = listen adr1 in
4      let m_04 = skI in
5      let m_05 = pkB in
6      let m_09 = Net.recv(c1) in
7      let m_13 = utf8("Alice") in
8      let m_15 = rsa_decrypt m_04 m_09 in
9      let (m_16,m_17) = iconcat m_15 in
10     let m_14 = m_17 in
11     let m_12 = concat m_13 m_14 in
12     let m_10 = rsa_encrypt m_05 m_12 in
13     let m_11 = Net.send c2 m_10 in
14     let m_20 = Net.recv(c2) in
15     let m_21 = Net.send c1 m_20 in
16     let m_22 = Net.recv(c1) in
17     let m_25 = rsa_decrypt m_04 m_22 in
```

```
18      let m_23 = rsa_encrypt m_05 m_25 in
19      let m_24 = Net.send c2 m_23 in
20      let z = Net.close c2 in 01
```

Listing 1.10: Role of the intruder in F$^{\$}$

7 Conclusion

We present BIFROST, a toolchain allowing to automatically generate `C` code from an abstract model of cryptographic protocols. BIFROST takes as inputs a protocol modeled in F$^{\$}$ and output `C` files. To generate these files BIFROST uses KaRaMeL to obtain an implementation of the protocol that corresponds to the model. Moreover, BIFROST transforms the F$^{\$}$ specifications into a π-calculus file using FS2PV and this file is sent to ProVerif to verify if the protocol is safe or not. If ProVerif finds a flaw, then we produce the additional `C` files that allow us to mount the attack on the protocol implementation. For this we have designed a parser of ProVerif's output, able to generate F$^{\$}$ model describing the attack trace. This F$^{\$}$ file can be translated into `C` code using KaRaMeL in the same way as other roles. Moreover, BIFROST can deal with several cryptographic primitives and network parameters. The choice of using ARM mBedTLS as a backend relies on its common use within industry products, especially within embedded systems. However, support with HACL*[6] which is a formally verified cryptographic library would be possible and make sense to have a completely verified toolchain. On a similar note, gcc/clang could be switched to CompCert[7], a formally verified compiler.

As a future work, we intend to switch FS2PV for an F* compatible translator that will allow us to support multiple verification tools alongside ProVerif. This will allow us to not rely on F# anymore and have a protocol representation only requiring to be compatible with F*. We also aim to extend BIFROST to be able to consider equational theories and advanced trace-based security properties like forward secrecy and post-compromise security. Security against side-channel and fault attacks could also be studied.

References

1. ARM mBed. `https://tls.mbed.org/`, accessed: 2022-01-21
2. OpenQuantumSafe. `https://openquantumsafe.org`, accessed: 2022-01-21
3. OpenSSL. `https://www.openssl.org/`, accessed: 2022-01-21
4. Python Lark parser. `https://lark-parser.readthedocs.io/en/latest/`, accessed: 2022-01-21
5. Cve-2008-0166 : Openssl 0.9.8c-1 (2008), `https://security-tracker.debian.org/tracker/CVE-2008-0166`

[6] `https://github.com/hacl-star/hacl-star`
[7] `https://compcert.org/`

6. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge university press (2005)
7. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1807–1823 (2017). https://doi.org/10.1145/3133956.3134078
8. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: Sok: Computer-aided cryptography. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 777–795. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00008, https://doi.org/10.1109/SP40001.2021.00008
9. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6841, pp. 71–90. Springer (2011). https://doi.org/10.1007/978-3-642-22792-9_5, https://doi.org/10.1007/978-3-642-22792-9_5
10. Basin, D., Mödersheim, S., Vigano, L.: An on-the-fly model-checker for security protocol analysis. In: European Symposium on Research in Computer Security. pp. 253–270. Springer (2003). https://doi.org/10.1007/978-3-540-39650-5_15
11. Bellare, M., Canetti, R., Krawczyk, H.: A modular approach to the design and analysis of authentication and key exchange protocols. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing. pp. 419–428 (1998). https://doi.org/10.1145/276698.276854
12. Benaissa, N., Méry, D.: Cryptographic protocols analysis in event b. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. pp. 282–293. Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_24
13. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(2), 1–45 (2011). https://doi.org/10.1145/1890028.1890031
14. Bhargavan, K., Bichhawat, A., Do, Q., Hosseyni, P., Küsters, R., Schmitz, G., Würtele, T.: Dy*: a modular symbolic verification framework for executable cryptographic protocol code. In: EuroS&P 2021-6th IEEE European Symposium on Security and Privacy (2021). https://doi.org/10.1109/EuroSP51992.2021.00042
15. Bhargavan, K., Corin, R., Deniélou, P.M., Fournet, C., Leifer, J.J.: Cryptographic protocol synthesis and verification for multiparty sessions. In: 2009 22nd IEEE Computer Security Foundations Symposium. pp. 124–140. IEEE (2009). https://doi.org/10.1109/CSF.2009.26
16. Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for tls. In: Proceedings of the 15th ACM conference on computer and Communications security. pp. 459–468 (2008). https://doi.org/10.1145/1455770.1455828
17. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(1), 1–61 (2008). https://doi.org/10.1145/1452044.1452049
18. Bieber, P., Boulahia-Cuppens, N., Lehmann, T., van Wickeren, E.: Abstract machines for communication security. In: 1993 Proceedings Com-

puter Security Foundations Workshop VI. pp. 137–146. IEEE (1993). https://doi.org/10.1109/CSFW.1993.246632
19. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001. pp. 82–96. IEEE (2001). https://doi.org/10.1109/CSFW.2001.930138
20. Blanchet, B.: A computationally sound mechanized prover for security protocols (2006). https://doi.org/10.1109/SP.2006.1, https://doi.org/10.1109/SP.2006.1
21. Ceelen, P., Mauw, S., Radomirović, S.: Chosen-name attacks: An overlooked class of type-flaw attacks. Electronic Notes in Theoretical Computer Science **197**, 31–43 (02 2008). https://doi.org/10.1016/j.entcs.2007.12.015
22. Cortier, V., Warinschi, B., Zălinescu, E.: Synthesizing secure protocols. In: European Symposium on Research in Computer Security. pp. 406–421. Springer (2007). https://doi.org/10.1007/978-3-540-74835-9_27
23. Cremers, C.J.: The scyther tool: Verification, falsification, and analysis of security protocols. In: International conference on computer aided verification. pp. 414–418. Springer (2008). https://doi.org/10.1007/978-3-540-70545-1_38
24. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c. In: International conference on software engineering and formal methods. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
25. Dierks, T., Rescorla, E.: The transport layer security (TLS) protocol version 1.2. RFC 5246, August (2008)
26. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on information theory **29**(2), 198–208 (1983). https://doi.org/10.1109/TIT.1983.1056650
27. Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., et al.: The matter of heartbleed. In: Proceedings of the 2014 conference on internet measurement conference. pp. 475–488 (2014). https://doi.org/10.1145/2663716.2663755
28. Escobar, S., Meadows, C., Meseguer, J.: Maude-npa: Cryptographic protocol analysis modulo equational properties. In: Foundations of Security Analysis and Design V, pp. 1–50. Springer (2009). https://doi.org/10.1007/978-3-642-03829-7_1
29. Katz, J., Yung, M.: Scalable protocols for authenticated group key exchange. In: Annual international cryptology conference. pp. 110–125. Springer (2003). https://doi.org/10.1007/978-3-540-45146-4_7
30. Lafourcade, P., Puys, M.: Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In: Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers. pp. 137–155 (2015). https://doi.org/10.1007/978-3-319-30303-1_9
31. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using fdr. In: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. pp. 147–166. Springer (1996). https://doi.org/10.1007/3-540-61042-1_43
32. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: International Conference on Computer Aided Verification. pp. 696–701. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_48
33. Modesti, P.: AnBx: Automatic generation and verification of security protocols implementations. In: International Symposium on Foundations and Practice of Security. pp. 156–173. Springer (2015). https://doi.org/10.1007/978-3-319-30303-1_10

34. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communications of the ACM **21**(12), 993–999 (1978). https://doi.org/10.1145/359657.359659
35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002). https://doi.org/10.1007/3-540-45949-9_5
36. Otway, D., Rees, O.: Efficient and timely mutual authentication. SIGOPS Oper. Syst. Rev. **21**(1), 8–10 (jan 1987). https://doi.org/10.1145/24592.24594, `https://doi.org/10.1145/24592.24594`
37. Pozza, D., Sisto, R., Durante, L.: Spi2Java: Automatic cryptographic protocol java code generation from spi calculus. In: Proceedings of the 18th International Conference on Advanced Information Networking and Application (2004). https://doi.org/10.1109/AINA.2004.1283943
38. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Béguelin, S.Z., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., et al.: Verified low-level programming embedded in f. Proc. ACM program. lang. **1**(ICFP), 17–1 (2017). https://doi.org/10.1145/3110261
39. Sprenger, C., Basin, D.: Developing security protocols by refinement. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 361–374 (2010). https://doi.org/10.1145/1866307.1866349
40. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. In: Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 387–398. PLDI '13 (2013). https://doi.org/10.1145/2499370.2491978